第七章 继承

面向对象程序设计(C++)

7 继承

7.1 继承与派生

7.2 继承方式

7.3 多继承

7.4 同名覆盖

7.5 构造函数与析构函数

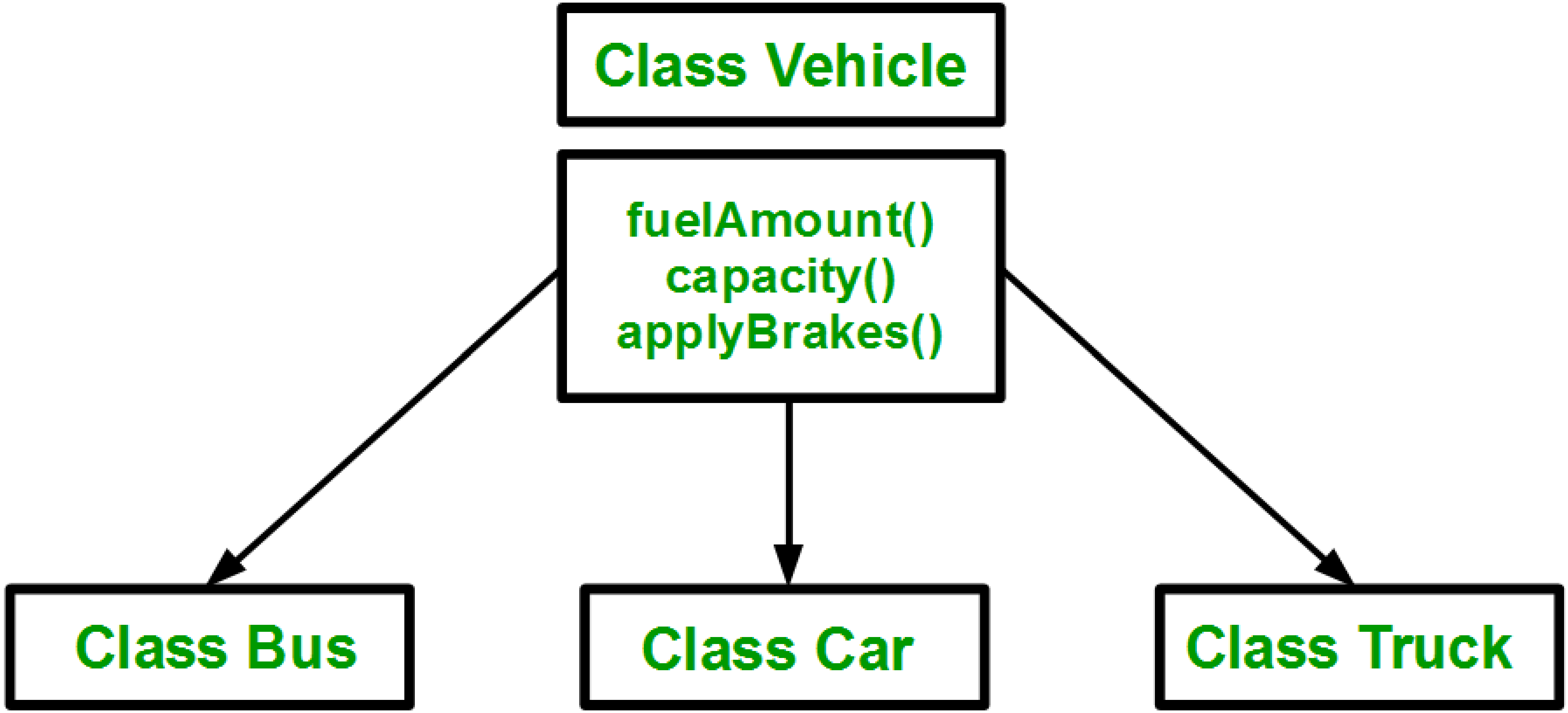
7.6 虚基类

7.7 类型转换

7.1 继承与派生

- **面向过程的程序设计中，需要为每一个项目单独进行一次程序开发，人们无法使用现有的软件资源**

- **面向对象技术强调软件的可重用性——继承机制，解决了软件重用问题**


Class Vehicle
fuelAmount()
capacity()
applyBrakes()
Class Bus
Class Car
Class Truck

Class Bus

fuelAmount()
capacity()
applyBrakes()

Class Car

fuelAmount()
capacity()
applyBrakes()

Class Truck

fuelAmount()
capacity()
applyBrakes()

- **继承**就是在一个已存在的类的基础上建立一个新的类

- 从已有的类(父类)产生一个新的子类，称为类的**派生**

- 一个基类可以派生出多个派生类，每一个派生类又可以作为基类再派生出新的派生类

例子

```
 class Student
{
public:
void display()
{cout<< “num: ” <<num<<endl;
cout<< “name: ” <<name<<endl;
cout<< “sex: ” <<sex<<endl;
}
private:
int num;
string name;
char sex;
};
```

例子

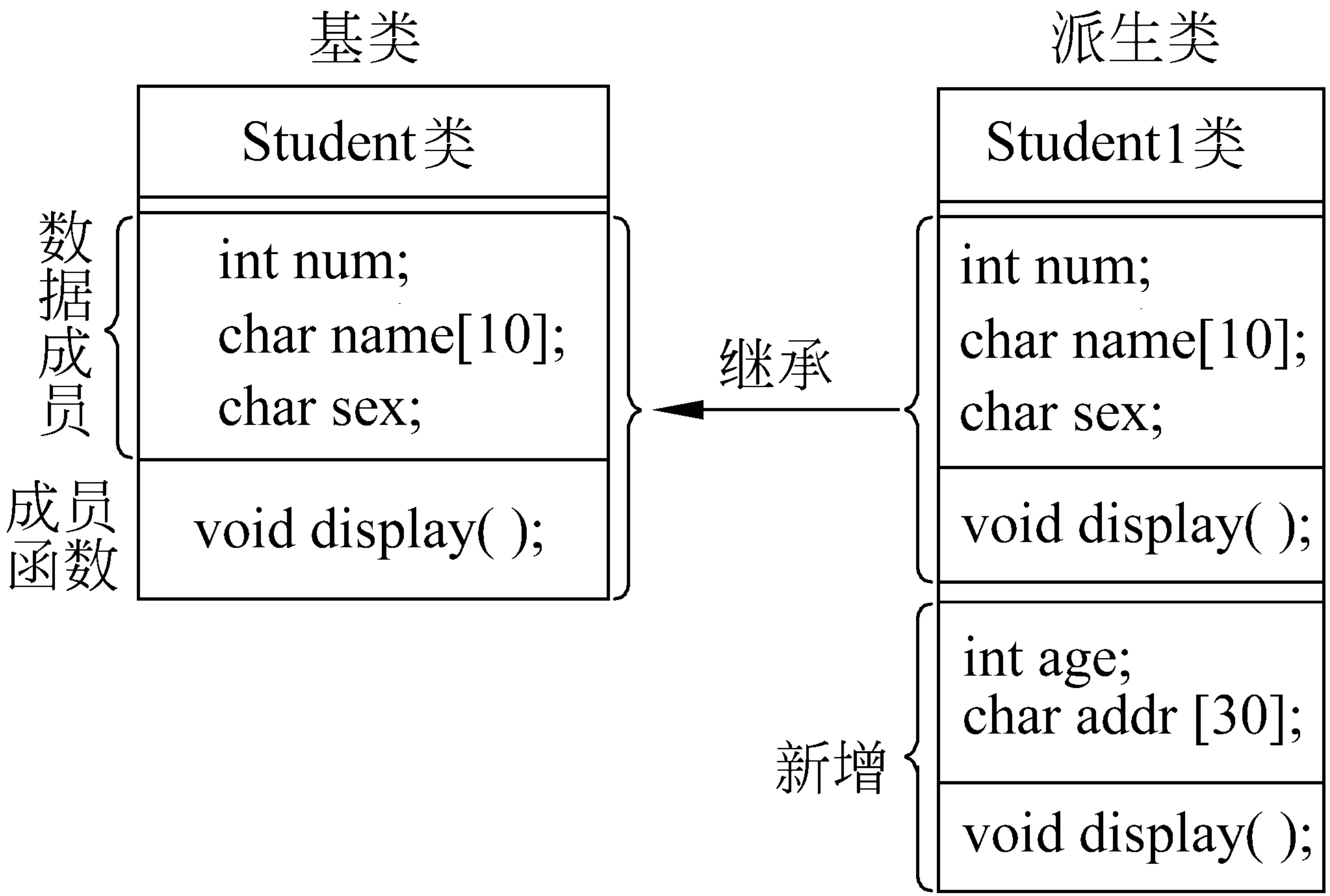
```
 class Student1    //  部分代码和功能可以重用
{
public:
void display()
{cout<< “num: ” <<num<<endl;
cout<< “name: ” <<name<<endl;
cout<< “sex: ” <<sex<<endl;
cout<< “age: ” <<age<<endl;
cout<< “address: ” <<addr<<endl;
}
private:
int num;
string name;
char sex;
int age;
char addr[20];
};
```

```
class Student1: public Student
{public:
void display_1( )
{cout<<"age: "<<age<<endl;
cout<<"address: "<<addr<<endl;}
private:
int age;
string addr;};
```

7.2 派生类的构成

派生类分为两部分:

- **基类继承来的成员**
- **声明派生类时增加的部分**



例子

```
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};
```

例子

```
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;

  Rect.setWidth(5);
  Rect.setHeight(7);

  cout << "Total area: " << Rect.getArea() << endl;
  //打印对象的面积
  return 0;
}
```

小结

- **可以声明一个基类，在基类中只提供某些基本功能，而另外的功能并未实现，然后在声明派生类时加入某些具体的功能，形成适用于某一特定应用的派生类。**

7.2 继承方式

class 派生类名: ［继承方式］ 基类名
{派生类新增加的成员} ;

```
class Derived : [virtual] [access-specifier] Base
{
  // member list
};
```

```
class Derived : [virtual] [access-specifier] Base1,
[virtual] [access-specifier] Base2, . . .
{
  // member list
};
```

派生类成员的访问属性

- 在建立派生类的时候，并不是简单地把基类的私有成员直接作为派生类的私有成员，把基类的公用成员直接作为派生类的公用成员
- 不同的继承方式决定了基类成员在派生类中的访问属性
- 类的默认继承方式是私有的

例子

```
class Person{
public:
    Person(const string& name, int age) : m_name(name), m_age(age){ }

    void ShowInfo()
    {
        cout << "姓名： " << m_name << endl;
        cout << "年龄： " << m_age << endl;
    }

protected:
    string  m_name;     //外部不可见

private:
    int     m_age;
};
```

例子

```
class Teacher : public Person{
public:
    Teacher(const string& name, int age, const string& title)
        : Person(name, age), m_title(title){}

    void ShowTeacherInfo()
    {
        ShowInfo();
        cout << "姓名： " << m_name << endl;
        cout << "年龄： " << m_age << endl;     //error

        cout << "职称： " << m_title << endl;
    }

private:
    string  m_title;
};
```

继承方式不影响对父类的访问权限

```
class Teacher : public  (protected, private) Person{
public:
    Teacher(const string& name, int age, const string& title)
        : Person(name, age), m_title(title){}

    void ShowTeacherInfo()
    {
        ShowInfo();
        cout << "姓名：" << m_name << endl;
        cout << "年龄：" << m_age << endl;    //error

        cout << "职称：" << m_title << endl;
    }

private:
    string  m_title;
};
```

Public继承

```
class Teacher : public Person
{
public:
    Teacher(const string& name, int age, const string& title)
        : Person(name, age), m_title(title) { }

    void ShowTeacherInfo()
    {
        ShowInfo();
        cout << "职称： " << m_title << endl;
    }

private:
    string  m_title;
};
```

Public继承

```
void TestPublic()
{
    Teacher teacher("李四", 35, "副教授");
    teacher.ShowInfo();
    cout << endl;
    teacher.ShowTeacherInfo();
}
```

Public继承

```
void TestPublic()
{
    Teacher teacher("李四", 35, "副教授");
    teacher.ShowInfo();
    cout << endl;
    teacher.ShowTeacherInfo();
}
```

姓名：李四
年龄：35

姓名：李四
年龄：35
职称：副教授

Private继承

```
class Teacher : private Person
{
public:
    Teacher(const string& name, int age, const string& title)
        : Person(name, age), m_title(title) {  }

    void ShowTeacherInfo()
    {
        ShowInfo();
        cout << "职称： " << m_title << endl;
    }

private:
    string  m_title;
};
```

Private继承

```
void TestPrivate()
{
    Teacher teacher("李四", 35, "副教授");
    teacher.ShowInfo();    //error
    cout << endl;
    teacher.ShowTeacherInfo();
}
```

●继承方式控制的是对象（用户）的访问权限

Protected继承

```
class Teacher : protected Person
{
public:
    Teacher(const string& name, int age, const string& title)
        : Person(name, age), m_title(title)  { }

    void ShowTeacherInfo()
    {
        ShowInfo();
        cout << "职称： " << m_title << endl;
    }

private:
    string  m_title;
};
```

Protected继承

```
void TestProtected()
{
    Teacher teacher("李四", 35, "副教授");
    teacher.ShowInfo();
    cout << endl;
    teacher.ShowTeacherInfo();
}
```

输出是什么?

Protected继承

```
void TestProtected()
{
    Teacher teacher("李四", 35, "副教授");
    teacher.ShowInfo();        //error
    cout << endl;
    teacher.ShowTeacherInfo();
}
```

Protected继承

```
class Leader : public Teacher
{
public:
   Leader(const string& name, int age, const string& title,
string position)
       : Teacher(name, age, title), m_position(position) {  }

   void ShowLeaderInfo()
   {
       ShowInfo();             //  right or wrong?
       ShowTeacherInfo();      // right or wrong?
       cout << m_position << endl;
   }

private:
   string m_position;
};
```

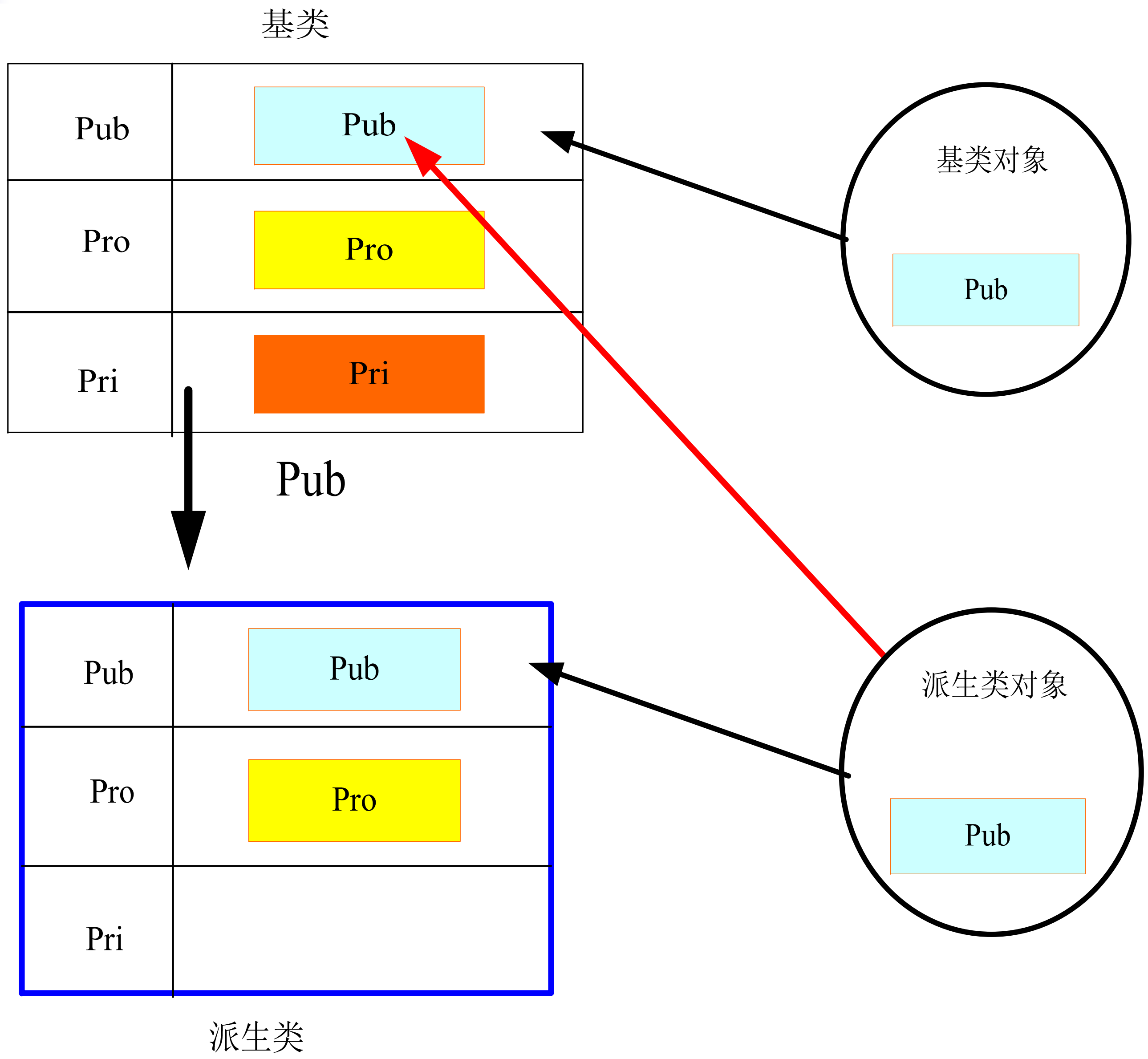
基类		继承方式	子类
public	&	public继承	public
public	&	protected继承	protected
public	&	private继承	private
protected	&	public继承	protected
protected	&	protected继承	protected
protected	&	private继承	private
private	&	public继承	子类无权访问
private	&	protected继承	子类无权访问
private	&	private继承	子类无权访问

子类不能访问基类私有成员，为什么?

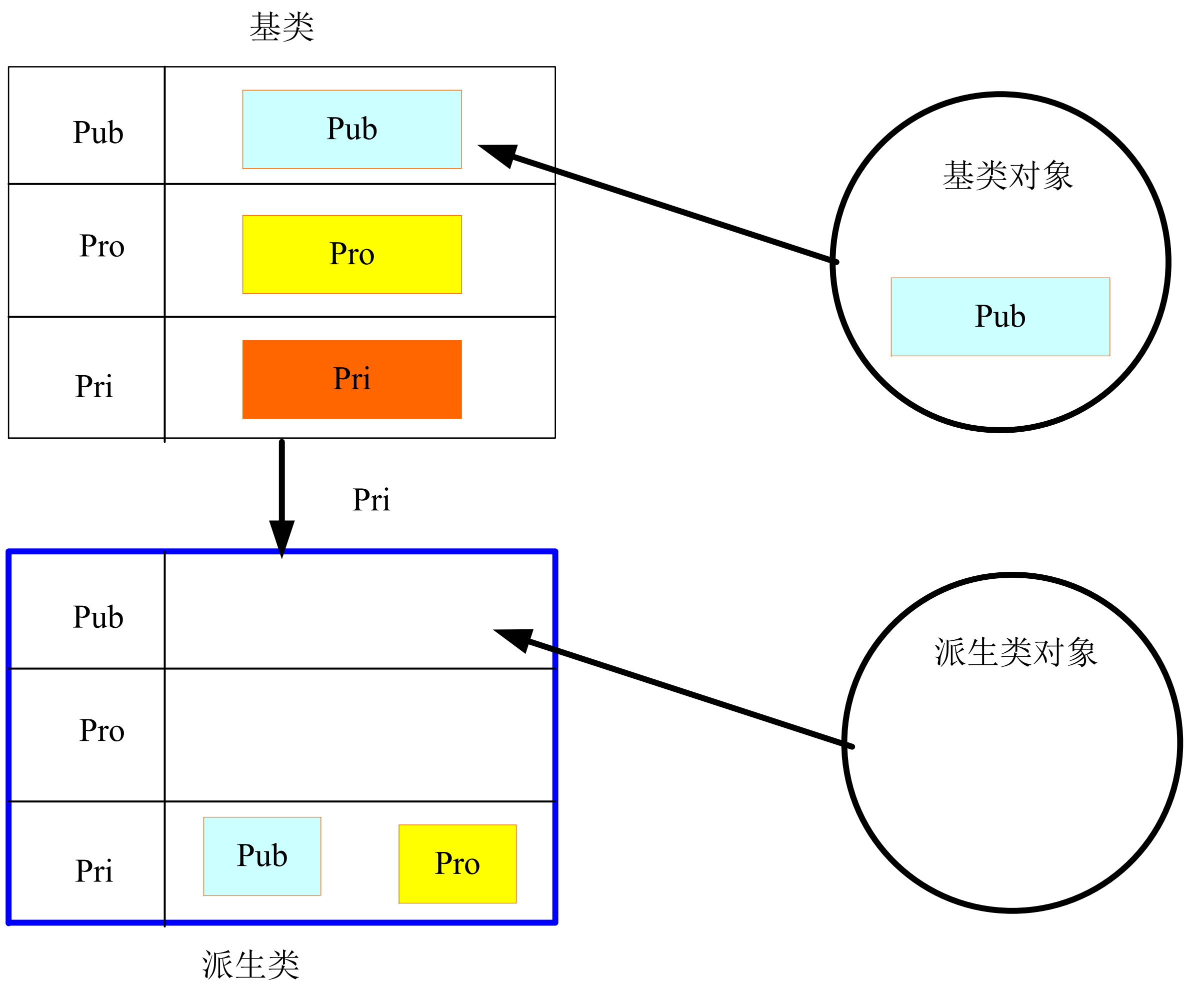
基类		继承方式	子类
public	&	public继承	public
public	&	protected继承	protected
public	&	private继承	private
protected	&	public继承	protected
protected	&	protected继承	protected
protected	&	private继承	private
private	&	public继承	子类无权访问
private	&	protected继承	子类无权访问
private	&	private继承	子类无权访问

子类不能访问基类私有成员，为什么?
(私有失效，继承即可访问)

Public继承


基类
Pub
Pub
Pro
Pro
Pri
Pri
基类对象
Pub
Pub
派生类对象
Pub
Pub
Pub
Pro
Pro
Pri
派生类

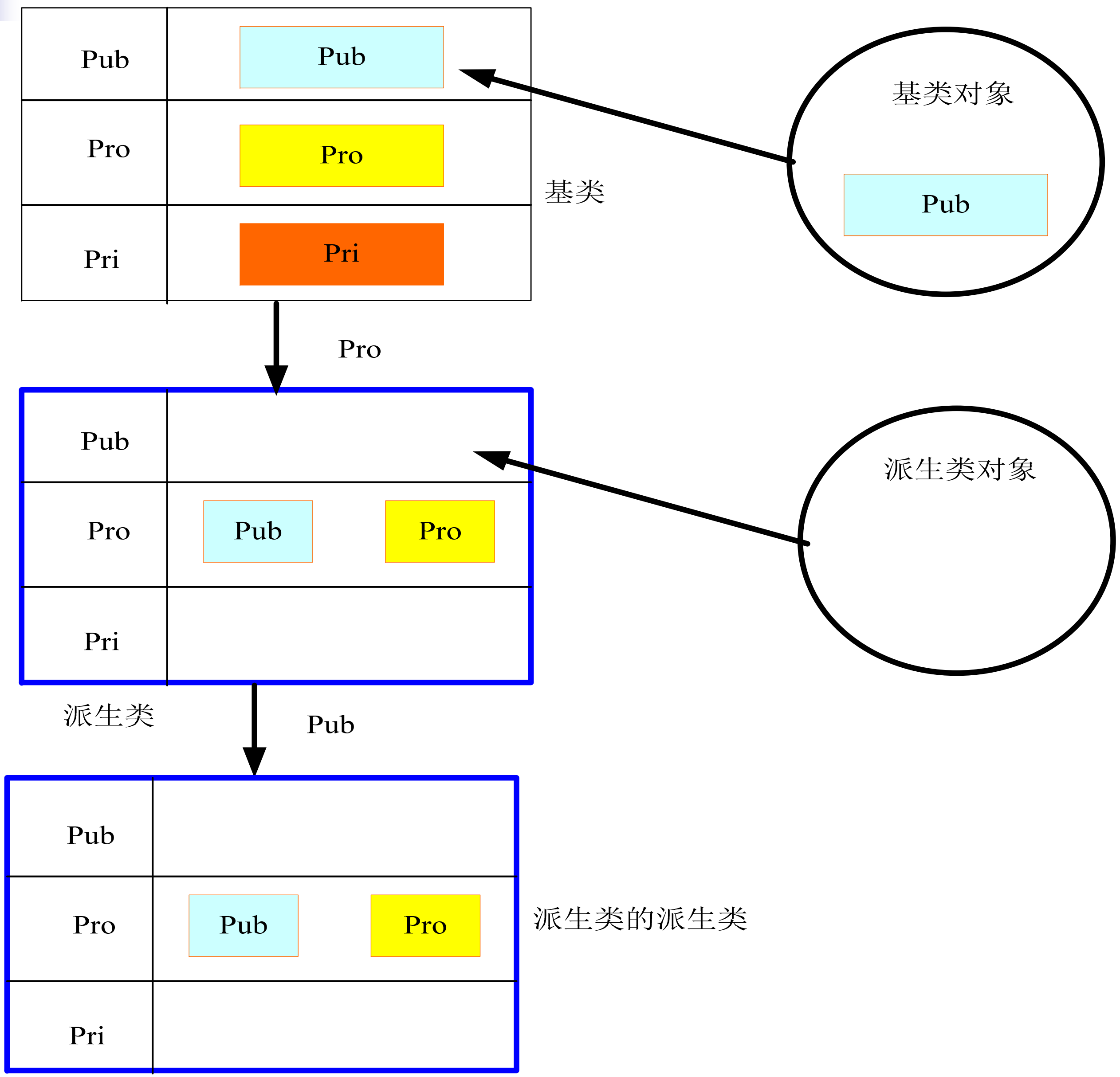
Private继承


基类
Pub
Pub
Pro
Pro
Pri
Pri
基类对象
Pub
Pri
Pub
Pro
Pri
Pub
Pro
派生类
派生类对象

Private继承

- 基类的成员只能由直接派生类访问，而不能再继承

Protected继承



例子1

```
class B1 {B1();};
class B2 {B2(int);};

class D1: public B1,B2 {

        D1(int i):B1{},B2{i} {}
        D1(int i):B2{i} {}
        D1(int i):B1{},B2{} {}
};
```

D1的初始化方式哪些是正确的?

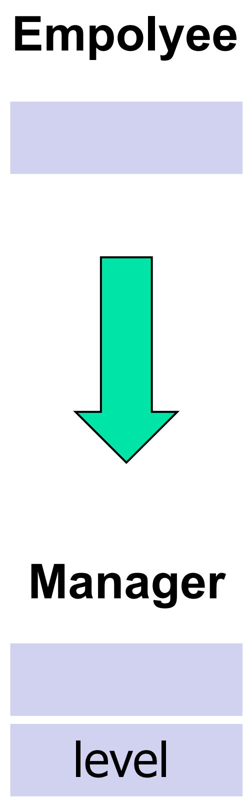
例子1

```
class B1 {B1();};
class B2 {B2(int);};

class D1: public B1,B2 {

        D1(int i):B1{},B2{i} {}
        D1(int i):B2{i}{}
        D1(int i):B1{},B2{} {}   //error
};
```

例子2

```
void g(Manage mm, Empolyee ee)
{
        Employee* pe=&mm;
        Manager*  pm=&ee;
        pe->level=2;
        pm=static_cast<Manager*>(pe);
        pm->level=2;
}
```

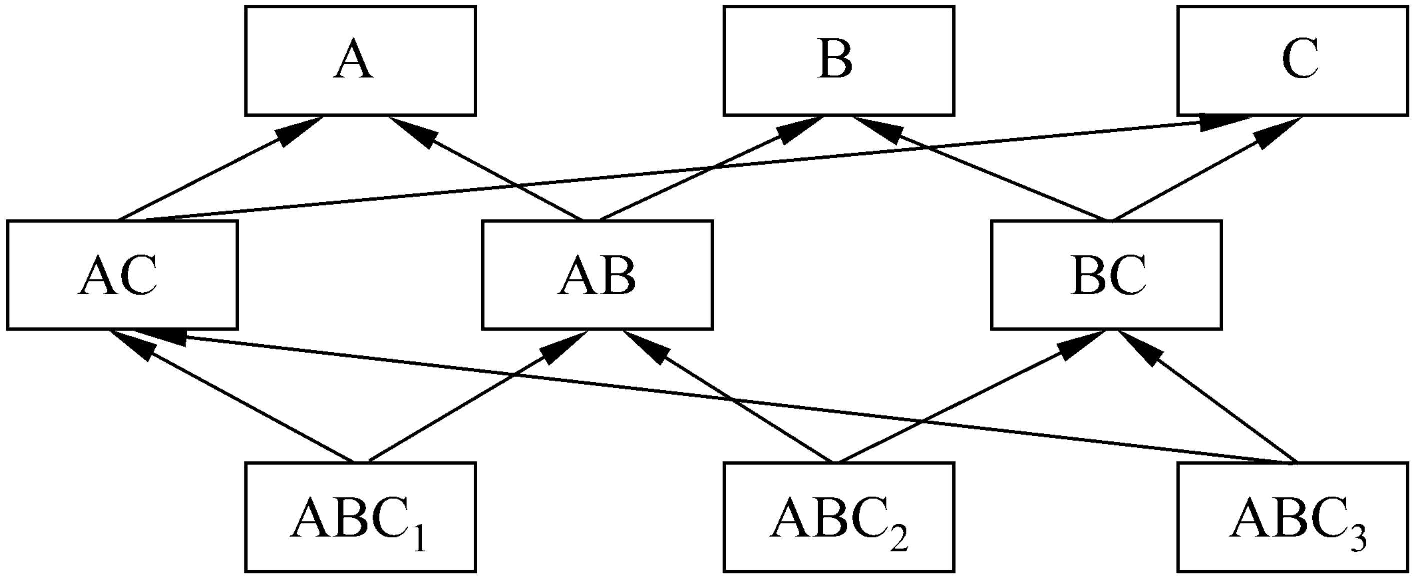
哪些代码是错误的?



例子2

```
void g(Manage mm, Empolyee ee)
{
      Employee* pe=&mm;
      Manager*  pm=&ee;   //error
      pe->level=2;   //error
      pm=static_cast<Manager*>(pe);
      pm->level=2;
}
```

7.3 多继承



```
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};

class PaintCost {
  public:
    int getCost(int area) {
      return area * 70;
    }
};
```

```
class Rectangle: public Shape, public PaintCost {
  public:
    int getArea() {    return (width * height);   }
};

int main(void) {
  Rectangle Rect;
  int area;

  Rect.setWidth(5);
  Rect.setHeight(7);

  area = Rect.getArea();

 cout << "Total area: " << Rect.getArea() << endl;

 cout << "Total paint cost: $" << Rect.getCost(area) << endl;

  return 0;
}
```

7.4 同名覆盖

```
class A
{
    public:

    void print2(){
        cout<<"A print2 !"<<endl;
    }
};


class B: public A
{

    public:

    void print2(int x){
        cout<<"B print2 !"<<x<<endl;
    }
};
```

```
int main(){
   B b;
   b.print2();    //error
   return 0;
}
```

- 编译器在作用域范围内查找函数名，如果找到了该函数名，编译器便停止查找，开始检查形参与实参的匹配是否合法， 如果不合法，不能通过编译。

```
int main(){
   B b;
   b.A::print2();;   //correct
   return 0;
}
```

另一种用法

```
class A
{
   public:
   void print2(){  cout<<"A print2 !"<<endl;  }
};

class B:public A
{
   public:
   using A::print2;
   void print2(int x){  cout<<"B print2 !"<<x<<endl; }
};

int main(){
   B b;
   b.print2();
   return 0;
}
```

7.5 构造函数与析构函数

- 构造函数不能继承
- 派生类的构造函数只负责对新增的成员进行初始化，对所有从基类继承来的成员，其初始化工作还是由基类的构造函数完成
- 如果基类没有声明构造函数，派生类也可以不声明构造函数，全部采用默认构造函数

```
class Parent
{
    public:

    Parent()
    {
        cout << "Inside base class" << endl;
    }
};

class Child : public Parent
{
    public:

    Child()
    {
        cout << "Inside sub class" << endl;
    }
};
```

```
int main() {

    Child obj;

    return 0;
}
```

Inside base class
Inside sub class

派生类初始化次序

1. 空间分配

2. 调用派生类构造函数

3. 派生类构造函数调用基类构造函数进行初始化。默认调用基类的默认构造函数（**const成员**）

4. 初始化列表进行初始化

5. 执行派生类构造函数的函数体

6. 返回

```
class Parent1
{
   public:

   Parent1()  {cout << "Inside first base class" << endl; }
};


class Parent2
{
   public:

   Parent2()  {cout << "Inside second base class" << endl; }
};

class Child : public Parent1, public Parent2 从左至右调用
{
   public:

   Child()  { cout << "Inside child class" << endl; }
};
```

```
int main() {

    Child obj1;
    return 0;
}
```

Inside first base class
Inside second base class
Inside child class

调用基类含参构造函数

```
class Parent{
   public:
   Parent(int i)
   {  int x =i;
      cout << "Inside base class's parameterised constructor" << endl;
   }
};

class Child : public Parent{
   public:
   Child(int j): Parent(j)
   {
      cout << "Inside sub class's parameterised constructor" << endl;
   }
};

int main() {
Child obj1(10);
   return 0;
}
```

多层继承

```
class  data{
        int d;
        public:
        data(int x){data::d = x;
        cout<<"class  data\n";}
};

class  A{
        data  d1;
        public:
        A(int x) : d1(x) {cout<<"class A\n";}
};

class   B: public A{
        data  d2;
        public:
        B(int x) : A(x),d2(x)  { cout<<"class B\n";}
};
```

```
class  C: public B{
        public:
        C(int  x) : B(x)   { cout<<"class C\n";}
};


int main( )
{
    C  object(5);
}
```

```
class  C: public B{
        public:
        C(int  x) : B(x)   { cout<<"class C\n";}
};


int main( )
{
   C  object(5);
}
```

```
class  data
class A
class  data
class B
class C
```

- 析构函数不能继承
- 执行派生类的析构函数时，系统会自动调用基类的析构函数和子对象的析构函数，对基类和子对象进行清理
- 调用的顺序与构造函数正好相反: 先执行派生类自己的析构函数，对派生类新增加的成员进行清理，然后调用子对象的析构函数，对子对象进行清理，最后调用基类的析构函数，对基类进行清理

Order of Inheritance

Class C (Base Class 2)

↓

Class B (Base Class 1)

↓

Class A (Derived Class)

Order of Constructor Call

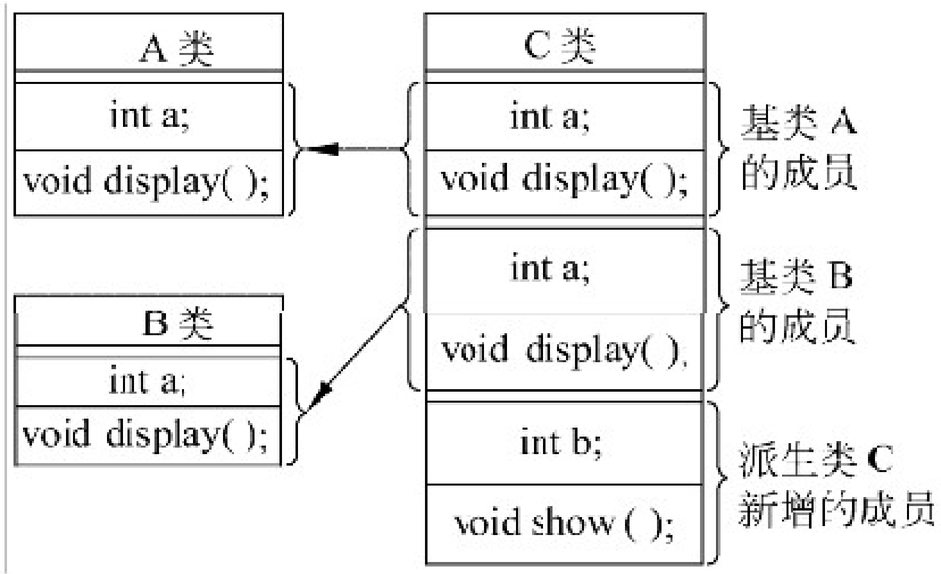
1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)

Order of Destructor Call

1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

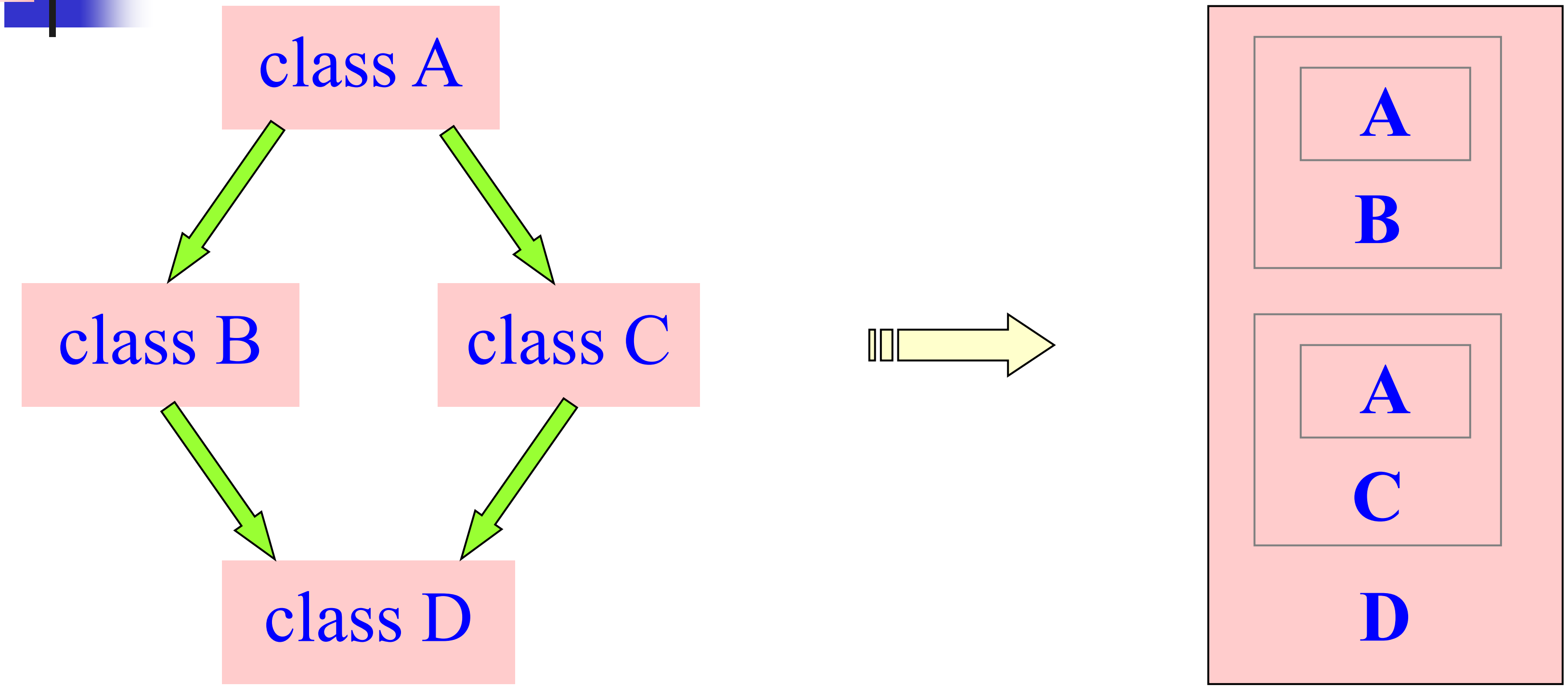
7.6 虚基类

- 多重继承中的二义性问题



c1.A::a=3

c1.A::display();



派生类D的对象中存在间接基类A的两份副本

解决方法：

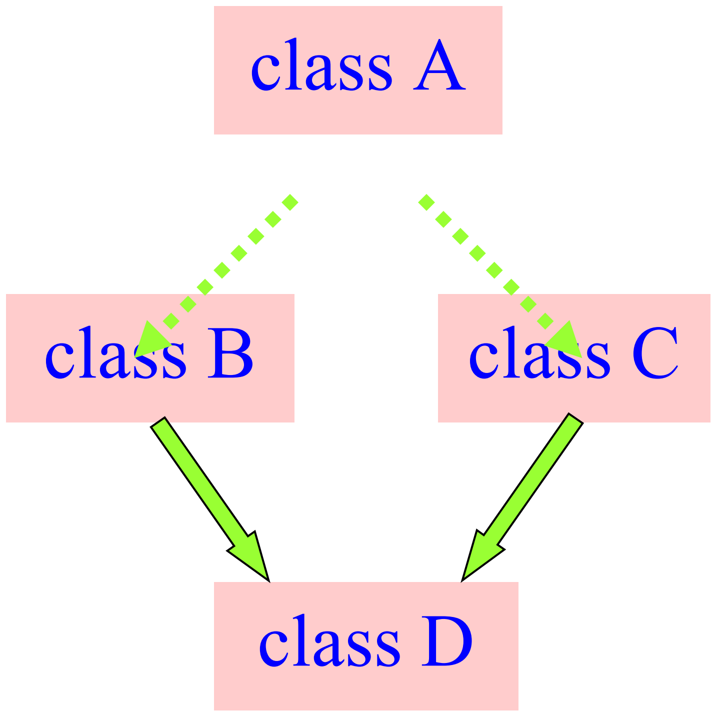
利用作用域限定符（::）把基类的成员与下一层基类关联起来：

d1.B::a=100; ← 从路径D→B→A继承而来

或：

d1.C::a=100; ← 从路径D→C→A继承而来

- 虚基类是一种派生方式（**virtual inheritance**）。类层次结构中虚基类的成员只出现一次，即基类的一个副本被所有派生类对象所共享



```
class  A
{
public:
     int  a;
};
```

```
class  B :  virtual  public  A
{
public:
     int  b;
};
```

```
class  C :  virtual  public  A
{
public:
     int  c;
};
```

```
D d1;
d1.a=1;  //无二义
```

虚基类的构造函数调用次序

```
class A
 {A(int i){ } …};

class B : virtual public A
{B(int n): A(n){ } …};

class C : virtual public A
{C(int n): A(n){ } …};

class D : public B, public C
{D(int n): A(n),B(n),C(n){ }…};
```

- 最后的派生类负责对虚基类初始化
- C++只执行最后的派生类对虚基类的构造函数的调用，而忽略虚基类的其他派生类(如类B和类C) 对虚基类的构造函数的调用

例子

```
class A{
public:
   A(){cout<<"class A"<<endl;}
};

class B: public A {
public:
   B(){cout<<"class B"<<endl;}
};

class C1:virtual public B{
public:
   C1(){cout<<"class C1"<<endl;}
};
```

例子

```
class C2:virtual public B{
public:
   C2(){cout<<"class C2"<<endl;}
};

class D:public C1, public C2 {
public:
   D(){cout<<"class D"<<endl;}
};

int main()
{
   D d;
   return 0;
}
```

例子

```
class C2:virtual public B{
public:
   C2(){cout<<"class C2"<<endl;}
};

class D:public C1, public C2 {
public:
   D(){cout<<"class D"<<endl;}
};

int main()
{
   D d;
   return 0;
}
```

例子

```
class C2:virtual public B{
public:
    C2(){cout<<"class C2"<<endl;}
};

class D:public C1, public C2 {
public:
    D(){cout<<"class D"<<endl;}
};

int main()
{
    D d;
    return 0;
}
```

```
class A
class B
class C1
class C2
class D
```

例子2

```
struct V{V(int i);};
struct A{A()};
struct B: virtual public V, virtual public A{B():V{1} {};};
class C: virtual public V{
  public:
          C(int i):V{i} {};
};
class D: virtual public B, virtual public C{
  public:
         D(){};
         D(int i):C{i} {};
         D(int i, int j):V{i},C{j} { };
};
```

画出继承的结构图。哪些D的构造函数是正确的，为什么?

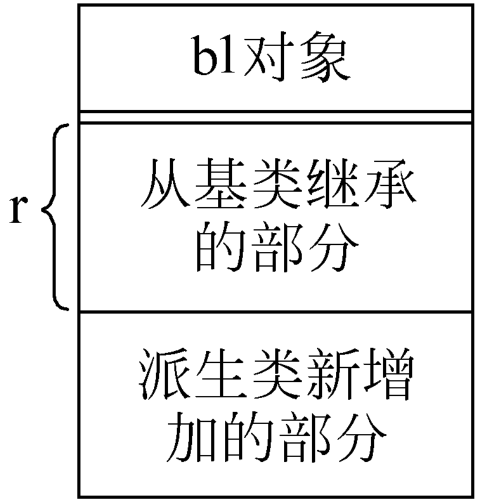
例子2

```
struct V{V(int i);};
struct A{A()};
struct B: virtual public V, virtual public A{B():V{1} {};};
class C: virtual public V{
   public:
            C(int i):V{i} {};
};
class D: virtual public B, virtual public C{
   public:
           D(){};  //error
           D(int i):C{i} {};  //error
           D(int i, int j):V{i},C{j} { };
};
```

初始化V是D的事！B，C有初始化V的代码，但并不唯一。而且初始化的顺序如何确定？

7.7 类型转换

```
student a1;
student1 b1;

student &r=b1;
```



- r不是b1的别名
- r是b1中基类部分的别名，r与b1中基类部分共享同一段存储单元
- r与b1具有相同的起始地址

继承中的拷贝构造函数

派生类到基类转型（默认）：问题容易出现在哪里？

预防措施：

（1）**delete**拷贝构造函数

（2）基类里定义为**private**或者**protected**